VirtuMatch: A Virtualized NL2SQL System for Distributed Job Recommendation.

Aman Sharma
Department of Computer Science
IIIT-Delhi
New Delhi, India
aman24013@iiitd.ac.in

Hariharan Iyer
Department of Computer Science
IIIT-Delhi
New Delhi, India
hariharan24038@iiitd.ac.in

Abstract

This project presents a Federated **NL2SQL** Job Recommender System, designed to bridge the gap between educational courses and job market requirements. The system allows users to query two distinct, independent MySQL databases one containing **software engineering job** data and the other **frontend engineering job** listings using natural language. By leveraging Large Language Models (LLMs) like **Llama-3 (via Groq)** and Gemini-Flash, the system parses natural language into specific SQL queries for separate schemas. A novel contribution of this work is "**Innovation:** ," an AI-driven ETL module that dynamically generates Python code to merge disparate database schemas on the fly based on sample data. The final output provides a unified, ranked, and summarized answer, enabling users to find relevant career paths and necessary upskilling courses seamlessly.

1 Introduction

In the modern recruitment ecosystem, efficiently mapping a candidate's skills and qualifications to suitable job roles is crucial. Traditional job portals rely on keyword-based search mechanisms, which often fail to understand the context of user queries. This project aims to bridge this gap by enabling Natural Language Query (NLQ) to SQL translation using a Large Language Model **(Gemini-Flash 2.5 API / Llama-3)** and executing it over distributed MySQL databases through a virtualization approach. The system automatically converts natural language inputs into executable SQL queries, fetches relevant results from integrated job databases, and presents them in a structured form.

2 Motivation and Objectives

2.1 Project Objectives

- **Federated Querying:** To enable natural language queries over two independent MySQL instances running on different IP address without physical data merging.
- **Intelligent Decomposition:** To break complex user questions into sub-queries **(software engineering job and frontend engineering job)** tailored to specific database schemas.
- **Dynamic Schema Alignment:**To automate the unification of differing table structures **(`Se_Job_Title` vs. `Fe_Job_Title`)** .
- **Resilient Execution:** To provide fallback mechanisms (Regex/Pattern-matching) when LLM inference fails or produces invalid SQL.

2.2 Motivation

The primary motivation is the "Closed-World Assumption" of traditional databases, where queries are limited to a single schema. In real-world scenarios, data resides in silos (e.g., LMS vs. ATS). Manually writing SQL Joins across these systems is often impossible due to network isolation or schema mismatches. This project uses an LLM as a "reasoning engine" to writing valid SQL for both systems and programmatically merge the results, simulating a unified view.

3 Data Sources and Schema

3.1 Data Sources:

The system connects to two distinct local MySQL instances:

- **software engineering job (Software Database):**Hosted on **IP Address IP1** . Contains data regarding software engineering roles, skills, and course details.
- **frontend engineer- ing job (Frontend Database):**Hosted on **IP address IP2**.

Contains data regarding frontend engineering jobs, salaries, and company profiles.

3.2 Database Schema

The system handles two highly similar but distinct schemas, differentiated primarily by their column prefixes to prevent ambiguity during federation:

3.2.1 Primary Database (Software Engineer Jobs):

- **Table:** `Software_engineer_jobs`
- **Columns:**
  - `se_Job_Id(PK)`
  - `se_Job_Title`
  - `se_skills`
  - `se_Experience`
  - `se_Salary_Range`
  - `se_location`
- **Schema Details:** Includes 18 columns such as `se_Qualifications`, `se_Benefits`, `se_Company_Size`.

3.2.2 Secondary Database (Frontend Engineer Jobs):

- **Table:** `frontend_engineer_jobs`
- **Columns:**
  - `fe_Job_Id (PK)`
  - `fe_Job_Title`
  - `fe_skills`
  - `fe_Experience`
  - `fe_Salary_Range`
  - `fe_location`
- **Schema Details:**Also contains 18 columns but uses the fe_ prefix. Unique values in fe_Preference (e.g., "Any", "Female", "Male") are used for filtering.

4 Methodology

4.1 Query Types:

The system handles various natural language inputs:

- **Topic-based queries:**e.g.,"Show me software jobs" (CourseDB) or "Find frontend developer roles" (JobDB).
- **Citation/Skill-focused queries:** e.g., "Jobs requiring React and TypeScript."
- **Temporal/Experience queries:** e.g., "Roles for candidates with 3 to 5 years of experience" .
- **Specific paper/role queries:** e.g., "Become a Data Scientist."

4.2 Query Processing Approach

- **LLM Query Rewriting:** The **Query-Analyzer.py** uses a system prompt to interpret user intent and generate JSON containing specific SQL and a "natural query" prompt.
- **Pattern-Based Decomposition:** If LLM fails, a fallback mechanism uses Regex (ROLE_EXTRACTION_PATTERN) to identify roles and generate standard SQL templates.
- **Intelligent SQL Generation:**Enforces strict MySQL syntax, specific table prefixes (se_ / fe_), and LOWER() for case-insensitive matching.
- **Federated Execution:** DatabaseExecutor manages separate connections to different IP adresses like IP1 and IP2, executing SQL independently.
- **Comprehensive Analysis:**Merged data is converted to Markdown and passed to a second LLM for final reasoning.

For complex queries (e.g., involving multiple roles or databases), the main query is split into sub-queries.

4.3 System Architecture: File Roles

The project follows a modular architecture designed for separation of concerns. The roles of the core files are described below:

- **query_analyzer.py**: Acts as the “Brain” of the pre-processing pipeline. It determines which databases are needed (`_needs_course_db`, `_needs_job_db`), constructs the structured prompt for the LLM to generate SQL, and handles the Regex fallback mechanism if the LLM fails to produce valid output.
- **executor.py**: Represents the “Power” of the system. It manages the connection pools to both MySQL ports (3306

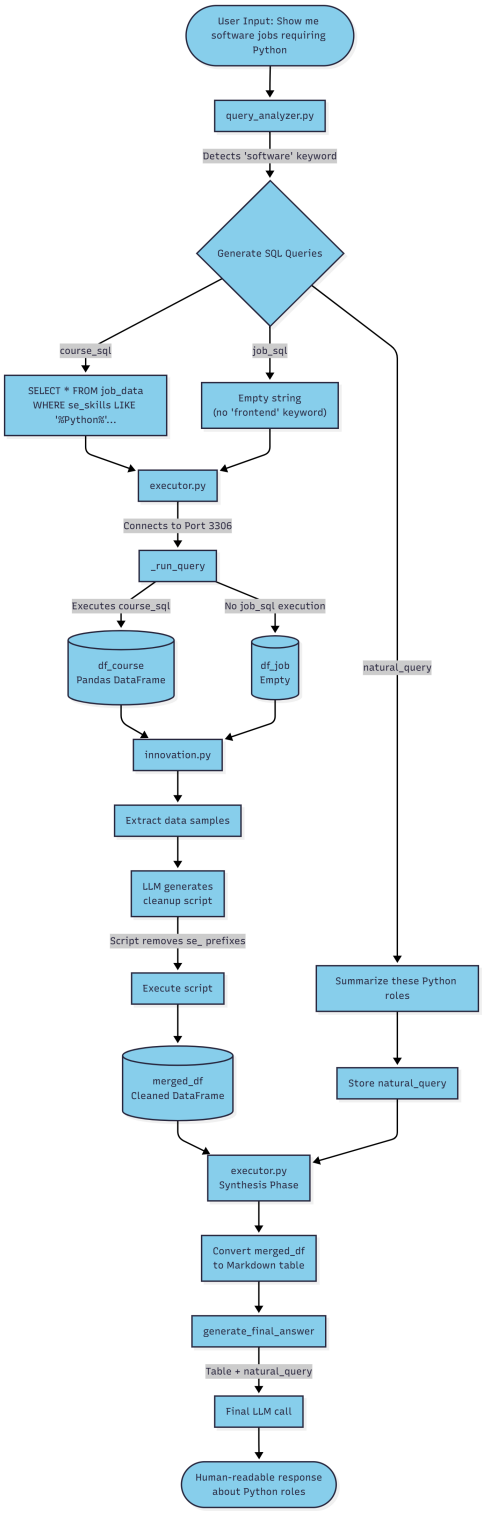

Figure 1: System Architecture Diagram illustrating the interaction between the User, Query Analyzer, Database Executor, and the Innovation Layer.

and 3307). It handles the safe execution of SQL queries and performs the final synthesis of the answer using the `generate_final_answer` function.

- **innovation1.py**: Serves as the "Bridge" between schemas. It implements the AI-driven ETL component. It extracts sample rows from the results, calls the LLM to generate a deterministic Python merge script, and executes that script in a secure sandbox to unify the two heterogeneous dataframes.
- **groq_client.py / gemeni_client.py**: The "Interface" layer to the AI models. These wrappers handle authentication and communication with **Llama-3** (via Groq) and **Gemini** APIs.

4.4 Query Processing Pipeline

- **Phase 1: Query Enhancement** The incoming query is analyzed for keywords. Lists such as COURSE_KEYWORDS (e.g., "software", "skills") and JOB_KEYWORDS (e.g., "frontend", "web") determine routing logic. If both domains are implied, both databases are flagged.
- **Phase 2:Structured Processing**
  - **Pattern-Based Decomposition:**Extracts roles and experience ranges (e.g., "5 years") using Regex to construct robust WHERE clauses.
  - **SQL Generation:**Constructs a prompt with schema summaries and "Mandatory DB Routing Rules."Output is strict JSON with course_sql and job_sql.
  - **Architecture Success Tracking:**Logs attempts/failures. If the LLM returns incomplete JSON, the system triggers deterministic SQL templates.
- **Phase 3:Federated Integration** This phase executes the queries and triggers Innovation to dynamically merge the results without hardcoded column mapping.

5 Hybrid Query Architecture:

The system employs a Hybrid Query Architecture that combines:

- **Structured Retrieval:**Precise SQL execution against relational databases to

fetch ground-truth data (salaries, skills, locations).

- **Unstructured Reasoning:** The natural_query generated by the analyzer instructs the final LLM on how to interpret the SQL results. For example, if SQL returns a list of skills, the natural_query might ask the LLM to "Compare these skills against current market trends," adding a layer of intelligence beyond simple data retrieval.

6 Core Algorithms

- **Federated SQL Planner (LLM-based):** Routes queries based on semantic context and enforces schema prefixes.
- **Dynamic ETL Generation (Innovation):**Uses LLM to analyze sample row data types and column names to write a deterministic merge script.
- **Regex Fallback Logic:** Deterministically parses "X to Y years" and role names to ensure reliability.

7 Conclusion

VirtuMatch successfully demonstrates that complex, cross-domain queries can be answered without centralizing data. By combining a robust SQL decomposition engine with AI-driven dynamic ETL (Innovation), the system creates a flexible "virtual" database view. This approach reduces the engineering overhead of maintaining manual schema mappings and allows for scalable integration of diverse data sources. The inclusion of regex fallbacks and hybrid structured/unstructured reasoning ensures the system remains reliable and valuable for end-users.

References

Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. Crowddb: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 61–72. ACM.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*.

Parker Glenn, Parag Pravin Dakle, Liang Wang, and Preethi Raghavan. 2024. Blendsql: A scalable dialect for unifying hybrid question answering in relational algebra. *arXiv preprint arXiv:2402.17882*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems*, volume 36.

Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, and Robert C. Miller. 2011. Crowdsourced databases: Query processing with people. In *CIDR 2011*.

Aditya Parameswaran and Neoklis Polyzotis. 2011. Answering queries using humans, algorithms and databases. *arXiv preprint*.

Hyunjung Park, Richard Pang, Aditya Parameswaran, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. 2012. Deco: A system for declarative crowdsourcing. In *Proceedings of the VLDB Endowment*.

Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2023. Querying large language models with sql. *arXiv preprint arXiv:2304.00472*.

Matthias Urban, Duc Dat Nguyen, and Carsten Binnig. 2023. Omniscientdb: a large language model-augmented dbms that knows what other dbmss do not know. In *Proceedings of the Sixth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–7.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Fuheng Zhao, Divyakant Agrawal, and Amr El Abbadi. 2024. Hybrid querying over relational databases and large language models. *arXiv preprint arXiv:2408.00884*.

Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. Db-gpt: Large language model meets database. *Data Science and Engineering*, 9(1):102–111.